INTEL CORPORATION

Vector Function Application Binary Interface

Version 0.9.5, January 22, 2013

Xinmin Tian, Hideki Saito
Sergey Kozhukhov, Kevin B. Smith, Robert Geva
Milind Girkar and Serguei V. Preis
Intel® Mobile Computing and Compilers

2013

3600 JULIETTE LANE, SANTA CLARA, CA 95054

1. Vector Function Overview

This section provides an overview of the **vector** functions (a.k.a named as **elemental** functions in Intel® Cilk™ Plus). The `vector` (or `simd` in OpenMP* syntax [3]) function annotation is applicable to C/C++ and Fortran functions. For syntax compatibility with the Microsoft compiler, it can appear in a __declspec; for compatibility with the GNU compiler, it can appear in an __attribute__ (for details see [1, 2]); in C++0X, it can appear in an *attribute-specifier*. It can also be specified with the OpenMP pragma syntax [3]. The syntax is

C/C++ syntax

__declspec(**vector**[(*[clause[[,] clause] ...]*)]) // Windows syntax
function definition or declaration

OR

__attribute__((**vector**[(*[clause[[,] clause] ...]*)])) // Linux syntax
function definition or declaration

OR

#pragma omp declare simd *[clause[[,] clause] ...]* // Windows and Linux syntax
function definition or declaration

Fortran syntax (Windows and Linux)

!dir$ attributes vector *[clause[[,] clause] ...]* :: one-*subroutine-or-function-name*

OR

!$omp declare simd(*one-subroutine-or-function-name***)** *[clause[[,] clause] ...]*

where the *clause* is one of the following:

- vectorlength(integer-constant-expression-list)
- linear(argument-list:linear-step)
- aligned(argument-list:alignment-constant)
- uniform(argument-list)
- mask
- nomask

The use of a "vector" annotation on a function declaration or definition enables the creation of vector versions of the function from the scalar version of the function that can be used to process multiple instances concurrently in a single invocation in a vector context (e.g. vectorizable loops). The maximum number of concurrent instances of the scalar function executed in a single instance of the vector function is determined by the vector-length used for the function. If the *vectorlength(…)* clause is used then vector-length corresponds to its specified value (or one of specified values) in the clause [1]. Otherwise, the value of vector-length is selected as described in Section 2.3. This ABI defines the details of the caller and callee interface of vector functions, including parameter passing and return value of vector functions.

2. Vector Function ABI

The vector function application binary interface (ABI) defines a set of rules that the caller and the callee functions must obey. These rules consist of

- Calling convention (how arguments are passed to the vector function and how values are returned from the vector function)
- Target processor and ISA class Selection
- Vector length (the number of concurrent scalar invocations to be processed per invocation of the vector function)
- Vector function masking
- Mapping from element data types to vector data types
- Vector function name mangling

2.1 Calling Convention

The calling convention defines the set of rules on how arguments are passed to a function and how the values are returned from the function. There are a number of calling conventions defined for IA-32 and Intel®64 architectures (for details see Appendix I). The vector functions must use the __regcall calling convention as described in the Appendix I, whether or not the original scalar function uses the __regcall calling convention. This can't be changed to use any other calling convention. However, the __regcall decoration (see __regcall Decoration sub-section in Appendix I) does not apply to vector function name mangling.

2.2 Default Target Processor and ISA Class Selection

This ABI defines several target processors and four ISA classes. For all the details of target processors and ISA classes, please refer to Section 3. The target processor can be explicitly specified with the `processor` clause as described in [1, 2], or the implicit default rule applies as follows.

For Intel® Xeon™ Phi™ (a.k.a. MIC) native and offload compilations, the default and the only supported target processor is "mic", and the selected ISA class is "mic".

For other IA-32 and Intel®64 processors, when the `processor` clause is not specified, the default target processor is the "pentium_4" for Windows* and Linux*, and the "pentium_4_sse3" for MacOS*; the ISA class for those target processors is "xmm". The only way to affect ISA class selection is through the `processor` clause. The command line processor flag has no impact on ISA class selection for the vector function ABI.

2.3 Vector Length

Every vector variant of a vector function has a vector length (VLEN). If the vectorlength clause is used, the VLEN is the value of the argument of that clause. The VLEN value must be power of 2 [1] .

If the `vectorlength` clause is not used, the notion of the function's "characteristic data type" is used to compute the vector length. The characteristic data type is defined in the following order:

a) For non-void function, the characteristic data type is the return type.
b) If the function has any non-uniform, non-linear parameters, then the characteristic data type is the type of the first such parameter.
c) If the characteristic data type determined by a) or b) above is struct, union, or class type which is pass-by-value (except for the type that maps to the built-in complex data type), the characteristic data type is `int`.
d) If none of the above three cases is applicable, the characteristic data type is int.
e) For Intel® Xeon™ Phi™ native and offload compilation, if the resulting characteristic data type is 8-bit or 16-bit integer data type, the characteristic data type is int.

The VLEN is then determined based on the characteristic data type and the selected ISA class's vector register (see Section 3 for target processor and ISA class selection in detail). The VLEN is computed using the formula below:

maximum_sizeof_ISA_Class_vector_register(*characteristic_data_type*) / *sizeof*(*characteristic_data_type*)

For example, if the target processor's ISA class is" xmm" and the characteristic data type of the function is int, the VLEN is 4.

2.4 Element Data Type to Vector Data Type Mapping

The vector data types for parameters are selected depending on ISA classes of target processors (see Section 3 for details), vector length, data type of original parameter, and parameter specification. For `uniform` and `linear` parameters, the original data type is preserved and they are passed by applying `__regcall` calling convention. For `vector` parameters, vector data types are selected by the compiler. The mapping from element data type to vector data type is described as below.

- The size of vector data type of parameter is computed as:
  size_of_vector_data_type = *VLEN* * sizeof(*original_parameter_data_type*) * 8 (in bits)
- The vector data type and number of vector registers are mapped based on the ISA class of the target processor. The vector data type is determined as described in the table 1, 2, 5 and 6. For instance, XMM ISA class uses XMM registers for integer data types. If VLEN = 4 and the parameter data type is "`int`", *size_of_vector_data_type* = 4 * 4 * 8 = 128 (bits), so the vector data type is MI128, which means one MI128 argument to be passed.
- If the *size_of_vector_data_type* is greater than the maximum width of the vector register supported by the ISA class of the target processor, multiple vector registers of the maximum size are selected and the parameter will be passed in multiple vector registers. For instance, XMM ISA class uses XMM registers for integer data types. If VLEN = 8 and the parameter data type is "`int`", *size_of_vector_data_type* = 8 * 4 * 8 = 256 (bits), so the vector data type is MI128, which means 2 MI128 arguments are to be passed.

The table 1 provides a set of mapping rules for all plain-old-data (POD) types on XMM ISA class. As shown in table 1, the rules for the vector data type selection of return value are the same as for vector parameters. Note that, the type "`long double`" is not supported in this Vector ABI. (‡) Note that the data type "`long`" is 64-bit `integer` on Intel®64 Linux* and MacOS* platforms; Otherwise, the "`long`" is 32-bit `integer`.

Type	Parameter / return value data type	sizeof (bytes)	XMM ISA Class (for Intel® SSE2/SSE3/SSSE3/SSE4.1/SSE4.2)			
			VLEN=2	VLEN=4	VLEN=8	VLEN=16
Integral	char /signed char unsigned char	1	1*MI128	1*MI128	1*MI128	1*MI128
	short / signed short unsigned short	2	1*MI128	1*MI128	1*MI128	2*MI128
	int / signed int unsigned int	4	1*MI128	1*MI128	2*MI128	4*MI128
	long / signed long unsigned long					
	long / signed long unsigned long‡	8	1*MI128	2*MI128	4*MI128	8*MI128
	__int64 unsigned __int64					
Pointer	any-type * any-type (*) ()	4 / 8	1*MI128/ 1*MI128	1*MI128/ 2*MI128	2*MI128/ 4*MI128	4*MI128/ 8*MI128
Floating-point	Float	4	1*MS128	1*MS128	2*MS128	4*MS128
	Double	8	1*MD128	2*MD128	4*MD128	8*MD128
	float complex	8	1*MS128	2*MS128	4*MS128	8*MS128
	double complex	16	2*MD128	4*MD128	8*MD128	16*MD128

Table 1: Element data type to vector data type mapping for XMM ISA Class

Type	Parameter / return value data type	sizeof (bytes)	Intel® Xeon™ Phi™			
			VLEN=2	VLEN=4	VLEN=8	VLEN=16
Integral	char / signed char unsigned char	1	Promote to int and apply int rule.			
	short / signed short unsigned short	2	Promote to int and apply int rule.			
	Int / signed int unsigned int	4	1*M512	1*M512	1*M512	1*M512
	long /signed long unsigned long					
	long / signed long unsigned long‡	8	1*M512	1*M512	1*M512	2*M512
	__int64 unsigned __int64					
Pointer	any-type * any-type (*) ()	4 / 8	1*M512/ 1*M512	1*M512/ 1*M512	1*M512/ 1*M512	1*M512/ 2*M512
Floating -point	float	4	1*M512	1*M512	1*M512	1*M512
	fouble	8	1*M512	1*M512	1*M512	2*M512
	float complex	8	1*M512	1*M512	1*M512	2*M512
	double complex	16	1*M512	1*M512	2*M512	4*M512

Table 2: Element data type to vector data type mapping for MIC target processors

The table 2 above provides the element data type to vector data type mapping rules for Intel® Xeon™ Phi™ processor (a.k.a MIC target).

2.4.1 Ordering of Vector Arguments

When a parameter in the original data type results in one argument in the vector function, the ordering rule is a simple one to one match with the original argument order. For example, when the original argument list is (int a, float b, int c), VLEN is 4, the ISA class is xmm, and all `a`, `b`, and `c` are classified `vector` parameters, the vector function argument list becomes (MI128 vec_a, MS128 vec_b, MI128 vec_c).

There are cases where a single parameter in the original data type results in the multiple arguments in the vector function. Those addition second and subsequent arguments are inserted in the argument list right after the corresponding first argument, not appended to the end of the argument list of the vector function. For example, the original argument list is (int a, float b, int c), VLEN is 8, the ISA class is xmm, and all `a`, `b`, and `c` are classified as `vector` parameters, the vector function argument list becomes (MI128 vec_a1, MI128 vec_a2, MS128 vec_b1, MS128 vec_b2, MI128 vec_c1, MI128 vec_c2).

2.5 Masking of Vector Function

For masked vector functions, the additional “mask” parameters are required. For non-MIC targets, each element of “mask” parameters has the data type of the characteristic data type (see Section 2.3). The number of mask parameters is the same as number of parameters required to pass the vector of characteristic data type for the given vector length. For example, with XMM targets, if characteristic data type is `int` and VLEN is 8, two MI128 mask parameters are passed (see table 2). The value of a mask parameter must be either bit patterns of all ones or all zeros for each element.

For the MIC target, the mask parameters are collection of 1-bit masks in unsigned integers. The total number of mask bits is equal to VLEN. The number of mask parameters is equal to the number of parameters for the vector of characteristic data type. The mask bits occupy the least significant bits of unsigned integer. For example, if the characteristic data type is double and VLEN is 16, there are 16 mask bits stored in two unsigned integers.

For each element of the vector, if the corresponding mask value is zero, the return value associated to that element is zero. Mask parameters are passed after all other parameters in the same order of parameters that they are apply to.

2.6 Vector Function Name Mangling

The name mangling of the generated vector function based on vector annotation serves as an important part of this ABI. It allows the caller and the callee functions to be compiled in separate files or compilation units. Using the function prototype in header files to communicate vector function annotation information, the compiler can perform function matching while vectorizing code at call sites. The vector function name is mangled as the concatenation of the following eight items as shown in the below vector function name decoration rules:

The descriptions of each item are:

- original_name - name of scalar function, including C++ and Fortran mangling
- vector_prefix - magic string, "_ZGV"
    - type_name is one char, 'l' for linear, 'u' for uniform , 'v' for vector.
    - For the `linear` parameter, the "stride" is string representation of the value of either positive integer constant stride or a 's' for variable stride uniform variable.
    - For the `uniform` or `vector` parameter, the "stride" is 1.

```
special-op::    // _Z
   ...
   'G' 'V' <isa> <mask> <vlen> <vparameters> '_' <original_name>

isa::
     'x'   // xmm  (SSE2)
   | 'y'   // ymm1 (AVX1)
   | 'Y'   // ymm2 (AVX2)
   | 'z'   // zmm  (mic)

<mask>::
      'M'  // mask
    | 'N'  // nomask

<vlen>::
   <decimal-number>

<vparameters>::
   /* empty */
   | <vparameter> <opt-align> <vparameters>

<vparameter>::
     's' decimal-number // linear, variable stride,
                        // decimal number is the position # of
                        // stride argument, which starts from 0
   | 'l' number // linear, constant stride
   | 'u'        // uniform
   | 'v'        // vector

<opt-align>::
   /* empty */
   | 'a' non-negative-decimal-number

<number> ::= [n] <non-negative decimal integer>   // n indicates negative
```

Given the example below with one annotation for the function "setArray", the compiler will generate two vector functions based on the scalar function "setArray". One is the vector function for XMM with masking, another one is the vector function for XMM without masking.

```
__declspec(vector(uniform(a), aligned(a:32), linear(k:1)))
extern float setArray(float *a, float x, int k)
{  a[k] = a[k] + x;
   return a[k];
}
```

For the above example, the mangled function names from vector annotations are shown below:

- **`_ZGVxN4ua32vl_setArray.`** -- without mask
- **`_ZGVxM4ua32vl_setArray.`** -- with mask

Where the "**_setArray.**" is the original mangled function name, "**_ZGV**' is the prefix of the vector function name, "**x**" indicates the xmm ISA class of the target processor, "**N**" indicates that this is a un-masked version", "**M**" indicates that this is a masked version, "**4**" is the vector length, "**ua32**" indicates uniform(a) and align(a:32), "**v**" indicates *private*(*x*) which is a default property for the argument of vector function, "**l**" indicates *linear*(*k*:1) – **k** is a linear variable whose stride is **1**.

3. `Processor` Clause for Changing ISA Class

The table 3 defines IA-32 and Intel®64 and Intel® Xeon™ Phi™ target processors, their ISA extensions and ISA classes.

Target processor	ISA extension	ISA class
pentium_4	SSE2	xmm
pentium_4_sse3	SSE3	xmm
core_2_duo_ssse3	SSSE3	xmm
core_2_duo_sse4_1	SSE4_1	xmm
core_i7_sse4_2	SSE4_2	xmm
core_2nd_gen_avx	AVX	ymm1
core_3rd_gen_avx	AVX	ymm1
core_4th_gen_avx	AVX2	ymm2
mic	Intel® Xeon™ Phi™ Instruction Set	mic

Table 3: Target processors, ISA extensions and ISA classes mapping

For the object level compatibility among compilers for IA-32 and Intel®64 based architectures, the compiler uses the selected ISA class to determine the caller / callee interface of the vector functions by following all rules described in this ABI documentation. The selection of default ISA class has been described in Section 2.2.

In order to achieve optimal performance for each processor target, the optional `processor` clause described in [1, 2] can be used for the compiler to generate vector function for YMM1 or YMM2 ISA class, which means the `processor` clause can be used to change ISA class from default XMM to YMM1 or YMM2. As described in Section 2.3, the ISA class affects vector length computation. For example:

- If the target processor is "core_2nd_gen_avx" or "core_3rd_gen_avx", their ISA class is "ymm1",
  - the VLEN is 4 if the characteristic data type of the function is `int`. (Integer vector operations in Intel® AVX are performed on `MI128`)
  - the VL EN is 8 if the characteristic data type of the function is `float`.
  - the VL EN is 4 if the characteristic data type of the function is `double`.
- If the target processor is "core_4th_gen_avx", its ISA class is "ymm2",
  - the VLEN is 8 if the characteristic data type of the function is `int`.
  - the VL EN is 8 if the characteristic data type of the function is `float`.
  - the VL EN is 4 if the characteristic data type of the function is `double`.

The table 4 and 5 provide the element data type to vector data type mapping rules of function parameter/return value of vector function interface for YMM1 and YMM2 ISA classes.

Type	Parameter / return value data type	sizeof (bytes)	YMM1 ISA Class (for Intel® AVX)			
			VLEN=2	VLEN=4	VLEN=8	VLEN=16
Integral	char / signed char unsigned char	1	1*MI128	1*MI128	1*MI128	1*MI128
	short / signed short unsigned short	2	1*MI128	1*MI128	1*MI128	2*MI128
	int / signed int unsigned int	4	1*MI128	1*MI128	2*MI128	4*MI128
	long / signed long unsigned long					
	long / signed long unsigned long‡	8	1*MI128	2*MI128	4*MI128	8*MI128
	__int64 unsigned __int64					
Pointer	any-type * any-type (*) ()	4 / 8	1*MI128/ 1*MI128	1*MI128/ 2*MI128	2*MI128/ 4*MI128	4*MI128/ 8*MI128
Floating-point	float	4	1*MS128	1*MS128	1*MS256	2*MS256
	double	8	1*MD128	1*MD256	2*MD256	4*MD256
	float complex	8	1*MS128	1*MS256	2*MS256	4*MS256
	double complex	16	1*MD256	2*MD256	4*MD256	8*MD256

Table 4: Element data type to vector data type mapping for YMM1 ISA Class

Type	Parameter / return value data type	sizeof (bytes)	YMM2 ISA class (for Intel® AVX2)			
			VLEN=2	VLEN=4	VLEN=8	VLEN=16
Integral	char / signed char unsigned char	1	1*MI128	1*MI128	1*MI128	1*MI128
	short / signed short unsigned short	2	1*MI128	1*MI128	1*MI128	1*MI256
	Int / signed int unsigned int	4	1*MI128	1*MI128	1*MI256	2*MI256
	long /signed long unsigned long					
	long / signed long unsigned long‡	8	1*MI128	1*MI256	2*MI256	4*MI256
	__int64 unsigned __int64					
Pointer	any-type * any-type (*) ()	4 / 8	1*MI128/ 1*MI128	1*MI128/ 1*MI256	1*MI256/ 2*MI256	2*MI256/ 4*MI256
Floating-point	float	4	1*MS128	1*MS128	1*MS256	2*MS256
	double	8	1*MD128	1*MD256	2*MD256	4*MD256
	float complex	8	1*MS128	1*MS256	2*MS256	4*MS256
	double complex	16	1*MD256	2*MD256	4*MD256	8*MD256

Table 5: Element data type to vector data type mapping for YMM2 ISA Class

Acknowledgement

The authors would like to thank Balaji V. Iyer (Intel) and Richard Henderson (Redhat) for his comments and feedback for this Vector Function ABI specification.

References

[1] Intel® Cilk™ Plus Language Extension Specification, Version 1.1, Document number: 324396-002US, http://software.intel.com/sites/default/files/m/4/e/7/3/1/40297-Intel_Cilk_plus_lang_spec_2.htm

[2] Tian, X., Saito, H., Preis, S.V., Kozhukhov, S.S., Cherkasov, A.G., Nelson, C., Panchenko, N., Geva, R.: Compiling C/C++ SIMD Extensions for Function and Loop Vectorization on Multicore-SIMD Processors. In Proc. of IEEE 26th International Parallel and Distributed Processing Symposium - Multicore and GPU Programming Models, Languages and Compilers Workshop, pp.2349 – 2358.

[3] Klemm, M. Duran, A., Tian, X., Saito, H., Caballero, D., Martorell, X.: Extending OpenMP* with Vector Constructs for Modern Multicore SIMD Architectures. IWOMP 2012: 59-72.

Appendix I

C/C++ Calling Conventions

This is from http://rathole.jf.intel.com/DOC/12_1_docs/lin/cpp/main_cls/index.htm

There are a number of calling conventions that set the rules on how arguments are passed to a function and how the values are returned from the function.

Calling Conventions on Windows* OS

The following table summarizes the supported calling conventions on Windows* OS:

Calling Convention	Compiler option	Description
__cdecl	`/Gd`	Default calling convention for C/C++ programs. Can be specified on a function with variable arguments.
__thiscall	none	Default calling convention used by C++ member functions that do not use variable arguments.
__clrcall	none	Calling convention that specifies that a function can only be called from managed code.
__stdcall	`/Gz`	Standard calling convention used for Win32 API functions.
__fastcall	`/Gr`	Fast calling convention that specifies that arguments are passed in registers rather than on the stack.
__regcall	`/Qregcall` ,which specifies that *__regcall* is the default calling	Intel Compiler calling convention that specifies that as many arguments as possible are passed in registers;

Calling Convention	Compiler option	Description
	convention for functions in the compilation, unless another calling convention is specified on a declaration	likewise, *__regcall* uses registers whenever possible to return values. This calling convention is ignored if specified on a function with variable arguments.

Calling Conventions on Linux* OS and Mac OS* X

The following table summarizes the supported calling conventions on Linux* OS and Mac OS* X:

Calling Convention	Compiler Option	Description
__attribute((cdecl))	none	Default calling convention for C/C++ programs. Can be specified on a function with variable arguments.
__attribute((stdcall))	none	Calling convention that specifies the arguments are passed on the stack. Cannot be specified on a function with variable arguments.
__attribute((regparm (number)))	none	On systems based on IA-32 architecture, the Intel 386, the regparm attribute causes the compiler to pass up to *number* arguments in registers EAX, EDX, and ECX instead of on the stack. Functions that take a variable number of arguments will continue to pass all of their arguments on the stack.
__regcall__attribute__((regcall))	`-regcall`, which specifies that *__regcall* is the default calling convention for functions in the compilation, unless another calling convention is specified on a declaration	Intel Compiler calling convention that specifies that as many arguments as possible are passed in registers; likewise, *__regcall* uses registers whenever possible to return values. This calling convention is ignored if specified on a function with variable arguments.

The __regcall Calling Convention

The *__regcall* calling convention is unique to the Intel compiler and requires some additional explanation.

To use __regcall, place the keyword before a function declaration. For example:

```
__regcall int foo (int i, int j);
```

`__attribute__((regcall)) foo (int I, int j);` (Linux OS and Mac OS X only)

Available __regcall Registers

All registers in a __regcall function can be used for parameter passing/returning a value, except those that are reserved by the compiler. The following table lists the registers that are available in each register class depending on the default ABI for the compilation. The registers are used in the order shown in the table.

Register class/Architecture	IA-32	Linux Intel® 64	Windows Intel® 64
GPR (see Note 1)	EAX, ECX, EDX, EDI, ESI	RAX, RCX, RDX, RDI, RSI, R8, R9, R12, R13, R14, R15	RAX, RCX, RDX, RDI, RSI, R8, R9, R10, R11, R12, R14, R15
FP	ST0	ST0	ST0
MMX	None	None	None
XMM	XMM0 - XMM7	XMM0 - XMM15	XMM0 - XMM15
YMM	YMM0 - YMM7	YMM0 - YMM15	YMM0 - YMM15

__regcall Data Type Classification

Parameters and return values for __regcall are classified by data type and are passed in the registers of the classes shown in the following table.

Type (for both unsigned and signed types)	IA-32	Intel® 64
bool, char, int, enum, _Decimal32, long, pointer	GPR	GPR
short, __mmask	GPR	GPR
long long, __int64	See Note 3; also see Structured Data Type Classification Rules	GPR
_Decimal64	XMM	GPR
long double	FP	FP
float, double, float128, _Decimal128	XMM	XMM
__m128, __m128i, __m128d	XMM	XMM

Type (for both unsigned and signed types)	IA-32	Intel® 64
__m256, __m256i, __m256d	YMM	YMM
__m512	n/a	n/a
complex type, struct, union	See Structured Data Type Classification Rules	See Structured Data Type Classification Rules

Note 2: For the purpose of structured types, the classification of GPR class is used.

Note 3: On systems based on IA-32 architecture, these 64-bit integer types (long long, __int64) get classified to the GPR class and are passed in two registers, as if they were implemented as a structure of two 32-bit integer fields.

Types that are smaller in size than registers than registers of their associated class are passed in the lower part of those registers; for example, float is passed in the lower 4 bytes of an XMM register.

__regcall Structured Data Type Classification Rules

Structures/unions and complex types are classified similarly to what is described in the x86_64 ABI, with the following exceptions:

- There is no limitation on the overall size of a structure.
- The register classes for basic types are given in Data Type Classifications.
- For systems based on the IA-32 architecture, classification is performed on four-bytes. For systems based on other architectures, classification is performed on eight-bytes.

__regcall Placement in Registers or on the Stack

After the classification described in Data Type Classifications and Structured Data Type Classification Rules, __regcall parameters and return values are either put into registers specified in Available Registers or placed in memory, according to the following:

- Each chunk (eight bytes on systems based on Intel® 64 architecture or four-bytes on systems based on IA-32 architecture) of a value of Data Type is assigned a register class. If enough registers from Available Registers are available, the whole value is passed in registers, otherwise the value is passed using the stack.
- If the classification were to use one or more register classes, then the registers of these classes from the table in Available Registers are used, in the order given there.
- If no more registers are available in one of the required register classes, then the whole value is put on the stack.

__regcall Registers that Preserve Their Values

The following registers preserve their values across a __regcall call, as long as they were not used for passing a parameter or returning a value:

Register class/ABI	IA-32	Linux Intel® 64	Windows Intel® 64
GPR	ESI, EDI, EBX, EBP, ESP	R12 - R15, RBX, RBP, RSP	R10 - R15, RBX, RBP, RSP
FP	None	None	None
MMX	None	None	None
XMM	XMM4 - XMM7	XMM8 - XMM15	XMM8 - XMM15
YMM	XMM4 - XMM7	XMM8 - XMM15	XMM8 - XMM15

All other registers do not preserve their values across this call.

__regcall Decoration

Function names used with *__regcall* are decorated. Specifically, they are prepended with *__regcall<n>__* before any further traditional mangling occurs. For example, the function `foo` would be decorated as follows: `__regcall3__foo`. This helps avoid improper linking with a name following a different calling convention, while allowing the full range of manipulations to be done with `foo` (such as setting a breakpoint in the debugger). The *<n>* part of the decoration specifies the version of the __regcall convention in affect (the current convention revision number is 3).